Kapitel 8

Lebendigkeitseigenschaften

Prof. Dr. Rolf Hennicker

30.06.2016

8.1 Der Begriff der Lebendigkeitseigenschaft

Lebendigkeitseigenschaften ("liveness properties") drücken aus, dass während der Ausführung eines parallelen Programms (irgendwann) "etwas Gutes" passiert.

Fortschrittseigenschaften sind spezielle Lebendigkeitseigenschaften ("progress properties"). Eine Fortschrittseigenschaft sichert zu, dass in jedem (fairen) Ablauf eines Programms ab jedem Zeitpunkt noch irgendwann eine spezifizierte Aktion ausgeführt wird.

Beispiel (Einspurige Brücke):

Ab jedem Zeitpunkt überquert irgendwann jedes wartende Auto die Brücke.

8.2 Fortschrittseigenschaften

Definition:

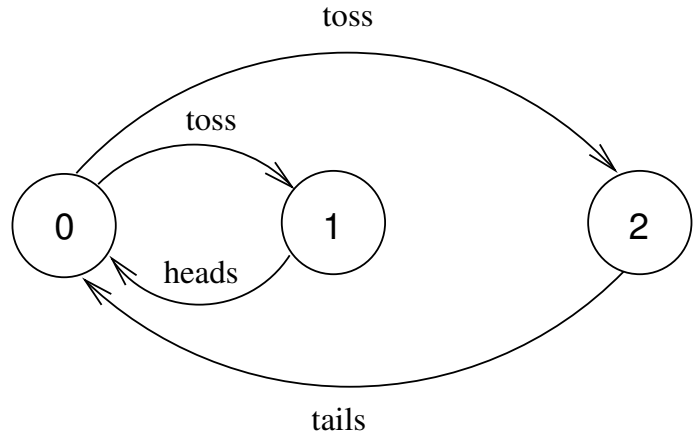
Sei F ein Name und sei $\{a_1, ..., a_n\} \subseteq$ Labels eine Menge von Aktionen ($n \geq 1$). Dann definiert

progress F = $\{a_1, ..., a_n\}$

eine *Fortschrittseigenschaft*. in FSP

Beispiel:

COIN = (toss → heads → COIN
| toss → tails → COIN).



progress HEADS = {heads}
progress TAILS = {tails}

COIN ⊨ HEADS ✓
COIN ⊨ TAILS ✓

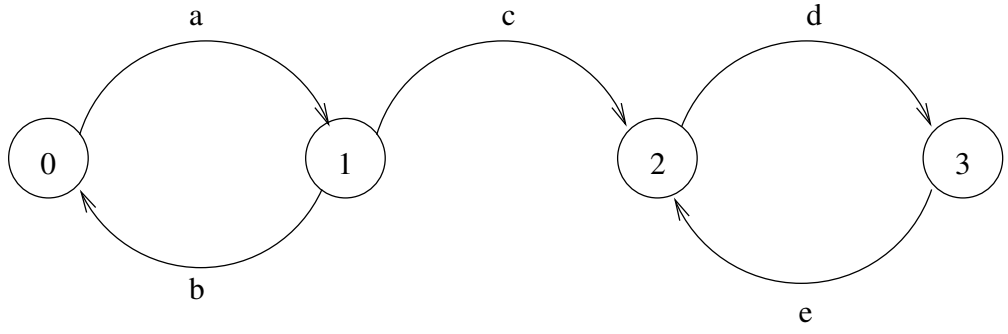
COIN erfüllt beide Fortschrittseigenschaften (HEADS und TAILS) unter der Annahme *fairer* Auswahl (von Alternativen).

Faire Auswahl:

Wenn während eines Ablaufs in einem LTS ein Zustand unendlich oft besucht wird, dann wird jede der zur Auswahl stehenden Transitionen immer wieder mal gewählt.

Fairer Ablauf: Ein fairer Ablauf ist ein Ablauf, der durch faire Auswahl entsteht.

Beispiel:



abababab ...
ist nicht fair, da im Zustand 1 irgendwann die Transition mit c gewählt werden müßte.

acdedede ...
ist fair, da Zustand 1 in diesem Ablauf nur einmal erreicht wird.

Bemerkung:

Jeder endliche Ablauf ist fair, weil kein Zustand unendlich oft besucht wird.

Generelle Voraussetzung:

Im Folgenden setzen wir immer faire Auswahl im LTS eines FSP-Prozesses voraus, d.h. wir betrachten bei der Überprüfung von Fortschrittseigenschaften nur faire Abläufe. (Unser Fairnessbegriff heißt auch "starke Fairness".)

Rechtfertigung:

Wir gehen davon aus, dass ein Betriebssystem eine faire Strategie der Prozessorzuteilung verwendet; zumindest solange es keinen guten Grund gibt davon abzuweichen (z.B um eine möglichst gute Prozessorauslastung zu erreichen; siehe Abschnitt 8.4).

Definition (Erfüllung von Fortschrittseigenschaften):

Sei Q ein Prozess und progress $F = \{a_1, \ldots, a_n\}$ eine Fortschrittseigenschaft.
Q *erfüllt* F, geschrieben $Q \models F$, wenn in jedem fairen Ablauf von Q mindestens eine Aktion aus $\{a_1, \ldots, a_n\}$ unendlich oft vorkommt.

Bemerkung:

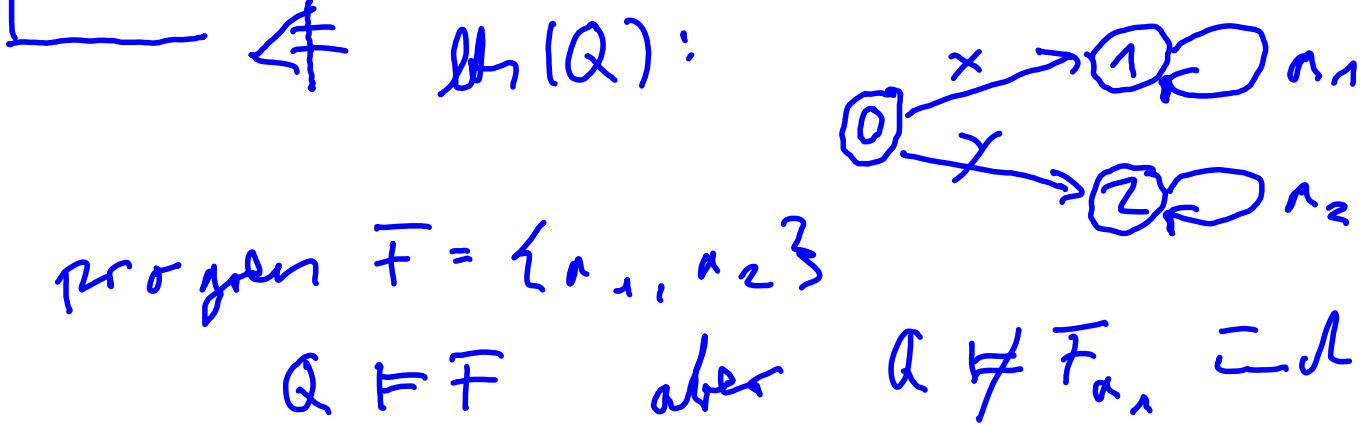
Sei progress $F = \{a_1, \ldots, a_n\}$ und progress $F_{a_i} = \{a_i\}$ für $i = 1 \ldots n$.

1. $(Q \models F_{a_i}$ für ein $i \in \{1, \ldots, n\}) \implies Q \models F$, aber nicht umgekehrt.
2. $(Q \models F_{a_i}$ für alle $i \in \{1, \ldots, n\}) \implies Q \models F$, aber nicht umgekehrt.

Bemerkung:

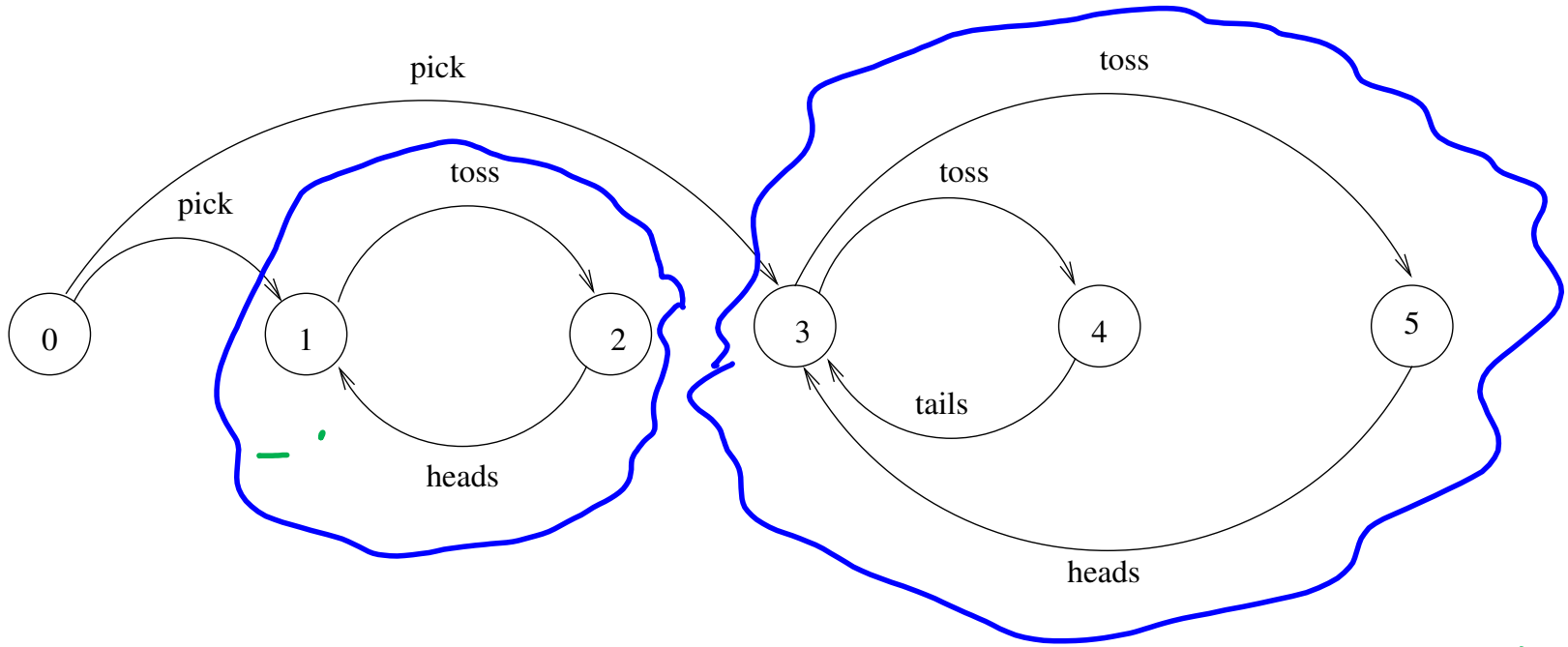
Ein Prozess Q, der ein Deadlock besitzt, erfüllt keine Fortschrittseigenschaft.

Begründung: Ein Ablauf zu einem Deadlock-Zustand ist endlich und damit fair. In diesem Ablauf kommt offensichtlich keine Aktion unendlich oft vor.



Beispiel (TWOCOIN):

$$\begin{aligned} \text{TWOCOIN} &= (\text{pick} \rightarrow \text{COIN} \mid \text{pick} \rightarrow \text{TRICK}), \\ \text{TRICK} &= (\text{toss} \rightarrow \text{heads} \rightarrow \text{TRICK}), \\ \text{COIN} &= (\text{toss} \rightarrow \text{heads} \rightarrow \text{COIN} \mid \text{toss} \rightarrow \text{tails} \rightarrow \text{COIN}). \end{aligned}$$



progress HEADS = {heads} wird von TWOCOIN erfüllt.
progress TAILS = {tails} wird von TWOCOIN nicht erfüllt.

Bemerkung:

Hätte man im LTS von TWOCOIN eine Transition von Zustand 2 zu Zustand 3, dann wäre TAILS erfüllt.

weil pick toss heads toss heads ... jetzt kein Kreis mit tails drin ist.

8.3 Nachweis von Fortschrittseigenschaften

Idee:

Suche "terminale" Mengen von Zuständen und überprüfe, welche Aktionen dort möglich sind.

Definition:

Sei Q ein Prozess mit $\text{lts}(Q) = (S, A, \Delta, q_0)$. Eine *terminale Menge* von Zuständen von Q ist eine nichtleere Teilmenge $T \subseteq S$, für die gilt:

1. Für alle $s \in T$ und für alle $(s, a, s') \in \Delta$ ist $s' \in T$ (d.h. T ist abgeschlossen unter Transitionen).
2. Jeder Zustand $s \in T$ ist von jedem Zustand $s' \in T$ erreichbar.

Folgerung:

In jedem fairen und unendlichen Ablauf w von Q, der in eine terminalen Menge T führt, wird die terminale Menge nicht mehr verlassen (wegen 1.) und jede Transition zwischen Zuständen von T wird unendlich oft durchlaufen (wegen 2. und weil w fair ist).

Beispiel (TWOCOIN):

- $\{1,2\}$ und $\{3,4,5\}$ sind die terminalen Mengen von Zuständen.
- $\{3,4\}$ ist nicht terminal, da (1) nicht erfüllt ist.
- $\{0,1,2\}$ ist nicht terminal, da (2) nicht erfüllt ist.
- $\{0,1,2,3,4,5\}$ ist nicht terminal, da (2) nicht erfüllt ist.

Satz 2:

Sei Q ein Prozess und sei progress F = $\{a_1,...,a_n\}$ eine Fortschrittseigenschaft. Q $\models$ F **genau dann, wenn** in jeder terminalen Menge T von Zuständen von Q (mindestens) eine Transition mit einer Aktion aus $\{a_1,...,a_n\}$ vorkommt, d.h. es gibt a $\in \{a_1,...,a_n\}$ und s, s' $\in$ T mit (s, a, s') $\in \Delta$.

Automatisches Checken von Fortschrittseigenschaften: Model-checking

1. Konstruiere alle terminalen Mengen von Zuständen im LTS von Q.
2. Falls es eine terminale Menge gibt, in der keine Transition mit einer Aktion aus $\{a_1,...,a_n\}$ vorkommt, wird F nicht von Q erfüllt; ansonsten wird F von Q erfüllt.

Beachte:

Die Gültigkeit einer Fortschrittseigenschaft ist entscheidbar, da es im LTS von Q nur endlich viele Zustände *und* endlich viele Transitionen gibt.

Beweisskizze für Satz 2:

O.E.d.A sei progress $F_a = \{a\}$.

1. Fall: Q hat ein Deadlock.

$\Rightarrow \exists$ endlichen Ablauf w von Q.

w endlich $\Rightarrow$ w fair.

Dann gilt $Q \not\models F_a$ und

$\exists$ terminale Menge, nämlich $T = \{z\}$ wobei z der Deadlock-Zustand ist, und $\not\exists\ z \xrightarrow{a} z$.

$\Rightarrow$ ($Q \models F_a \Leftrightarrow$ in jeder terminalen Menge...)

$$[\ \neg x \wedge \neg y \Rightarrow \underbrace{(x \Leftrightarrow y)}$$

$$(\neg x \vee y) \wedge (\neg y \vee x)\ ]$$

2. Fall: Q hat kein Deadlock.

„⇐": Sei w ein beliebiger fairer Ablauf von Q.

2. Fall ⇒ w ist unendlich.

Weil lts(Q) endlich ist, gibt es einen Zustand z_1 im lts(Q), der von w unendlich oft besucht wird.

Betrachte $T = \{z \in \hat{S} \mid z$ wird von w von z_1 aus besucht$\}$ (Zustände von lts(Q))

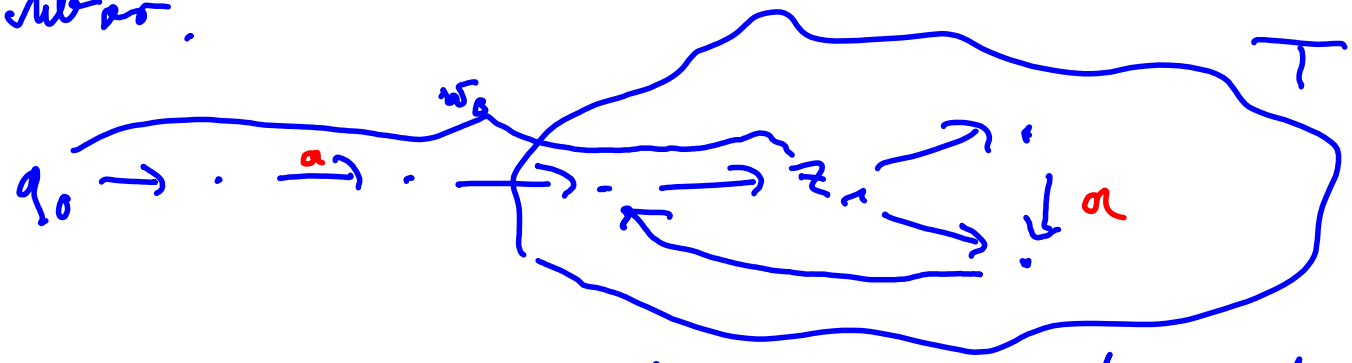
T ist eine terminale Menge.

Voraussetzung: ∃ Transition $z' \xrightarrow{a} z''$ mit $z', z'' \in T$.

Weil T terminal ist und weil w fair ist, wird diese Transition von w unendlich oft durchlaufen.

⇒ a kommt in w unendl. oft vor ⇒ $Q \models F_a$

"$\Rightarrow$": Sei T eine beliebige terminale Menge von Q.
Sei $z_1 \in T$. z_1 ist, weil $lts(Q)$ erreichbar ist,
vom Anfangszustand q_0 durch eine endliche Aktionsfolge
w_0 erreichbar.



Da T eine terminale Menge ist und z_1 kein
Deadlock-Zustand sein kann, kann w_0 zu einem
unendlichen fairen Ablauf w fortgesetzt werden,
der nur Zustände aus T besucht.
Vor.: $Q \models F \Rightarrow a$ kommt in w unendl. oft vor.
$\Rightarrow \exists$ Transition zwischen Zuständen in T
mit der Aktion a. $\square$

Beispiel (TWOCOIN):

- Terminale Mengen sind $T_1 = \{1, 2\}$, $T_2 = \{3, 4, 5\}$.
- In T_1 und in T_2 gibt es eine Transition mit "heads".
  Also ist HEADS erfüllt.
- In T_1 gibt es keine Transition mit "tails".
  Also ist TAILS nicht erfüllt.

Default-Analyse:

Für alle Aktionen a im Alphabet eines Prozesses Q wird überprüft, ob

$$\text{progress } \mathsf{F}_a = \{\mathsf{a}\}$$

von Q erfüllt wird.

Beispiel (Brücke):

Die Default-Analyse zeigt, dass alle Fortschrittseigenschaften erfüllt sind.
In jedem *fairen* Ablauf wird jedes Auto unendlich oft über die Brücke fahren.

Aber: Beim Programmlauf mit 3 Autos jeder Farbe überqueren nur Autos einer Farbe die Brücke.
Grund: Das Betriebssystem sorgt für eine möglichst hohe Auslastung des Prozessors, wodurch es zu bestimmten unfairen Abläufen kommen kann. Diese wurden in unserer Analyse bisher nicht erfasst.

8.4 Aktionsprioritäten

Zur Programmlaufzeit können unter bestimmten Bedingungen auch gewisse unfaire Abläufe vorkommen. Dann ist Fortschritt nicht mehr gesichert, auch wenn alle Fortschrittseigenschaften bewiesen wurden.

Aktionsprioritäten dienen dazu, solche Situationen zu modellieren.

Idee:

Die "gewissen" unfairen Abläufe werden im Modell künstlich zu fairen Abläufen gemacht. Das System wird unter "Stress" gesetzt.
(Z.B. möglichst viele Autos derselben Farbe gleichzeitig auf die Brücke).

Die Fortschrittseigenschaften werden unter dem gestressten Systemmodell erneut untersucht. Falls sie nicht mehr gelten, muss das Modell und, falls bereits implementiert, auch das Programm geeignet modifiziert werden.

Prozesse mit Aktionsprioritäten

Definition:

Sei E ein Prozessausdruck und $a_1,\ldots,a_n \in \alpha E$.

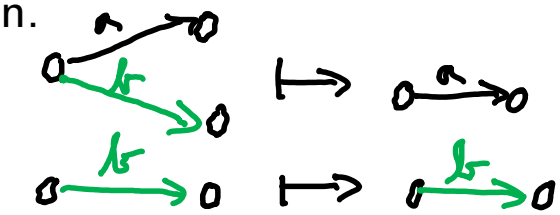
1. **Hohe Priorität:**

   $(E) << \{a_1,\ldots,a_n\}$

   ist ein Prozessausdruck, in dem die Aktionen $a_1,\ldots,a_n$ *hohe Priorität* haben.

   Wirkung:

   Wo immer eine Auswahl im LTS von E vorkommt zwischen $a \in \{a_1,\ldots,a_n\}$ und $b \notin \{a_1,\ldots,a_n\}$ wird die Transition von b weggelassen.



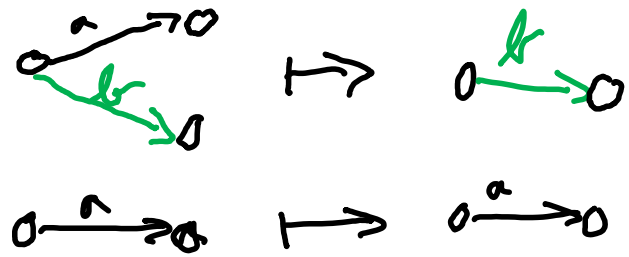
2. **Niedrige Priorität:**

   $(E) >> \{a_1,\ldots,a_n\}$

   ist ein Prozessausdruck in dem die Aktionen $a_1,\ldots,a_n$ *niedrige Priorität* haben.

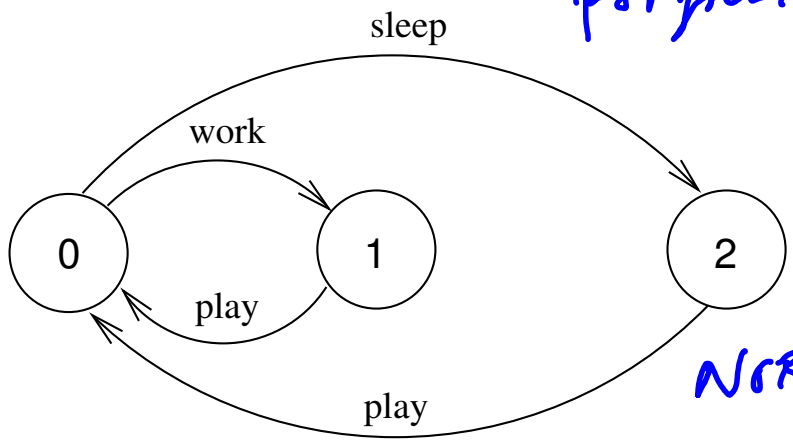
   Wirkung:

   Wo immer eine Auswahl im LTS von E vorkommt zwischen $a \in \{a_1,\ldots,a_n\}$ und $b \notin \{a_1,\ldots,a_n\}$ wird die Transition von a weggelassen.

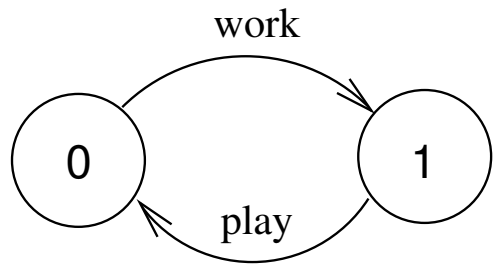


Beispiel:

$$\text{NORMAL} = (\text{work} \rightarrow \text{play} \rightarrow \text{NORMAL} \\ |\text{sleep} \rightarrow \text{play} \rightarrow \text{NORMAL}).$$



$\|\text{WORKOHOLIC} = (\text{NORMAL}) << \{\text{work}\}.$



Semantik von Aktionsprioritäten

Sei E ein Prozessausdruck und $a_1,\ldots,a_n \in \alpha E$.

Sei $lts(E) = (S, A, \Delta, q_0)$.

Hohe Priorität:

$$lts(E<<\{a_1,\ldots,a_n\}) =_{def} Reach(S, A, \Delta_{<<}, q_0),$$

wobei $\Delta_{<<} =$

$$\{(q, a, q') \in \Delta \mid a \in \{a_1, \ldots, a_n\} \cup$$
$$\{(q, b, q') \in \Delta \mid b \notin \{a_1, \ldots, a_n\} \text{ und } \nexists (q, a, q') \in \Delta \text{ mit } a \in \{a_1, \ldots, a_n\}\}.$$

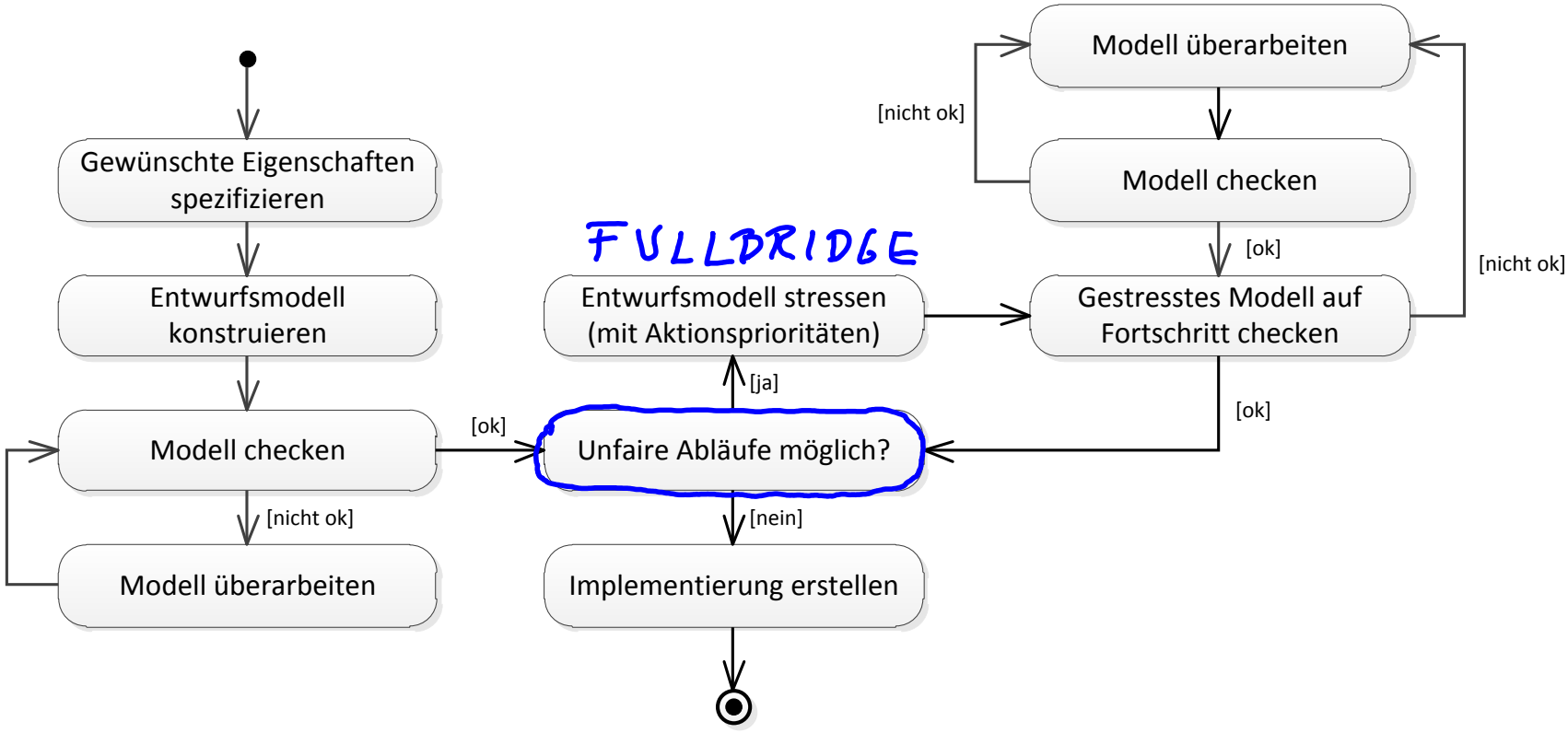
Niedrige Priorität:

$$lts(E>>\{a_1,\ldots,a_n\}) =_{def} Reach(S, A, \Delta_{>>}, q_0),$$

wobei $\Delta_{>>} =$

$$\{(q, b, q') \in \Delta \mid b \notin \{a_1, \ldots, a_n\} \cup$$
$$\{(q, a, q') \in \Delta \mid a \in \{a_1, \ldots, a_n\} \text{ und } \nexists (q, b, q') \in \Delta \text{ mit } b \notin \{a_1, \ldots, a_n\}\}.$$

Entwicklungsmethodik unter Einbeziehung von Aktionsprioritäten



Beispiel (Brücke):

Sicherheitseigenschaften und Deadlock-Freiheit wurden schon gecheckt.

Die Defaultanalyse zeigt, dass alle Fortschrittseigenschaften erfüllt sind.

Welche unfairen Abläufe können in der Praxis (d.h. beim Programmablauf) auftreten?

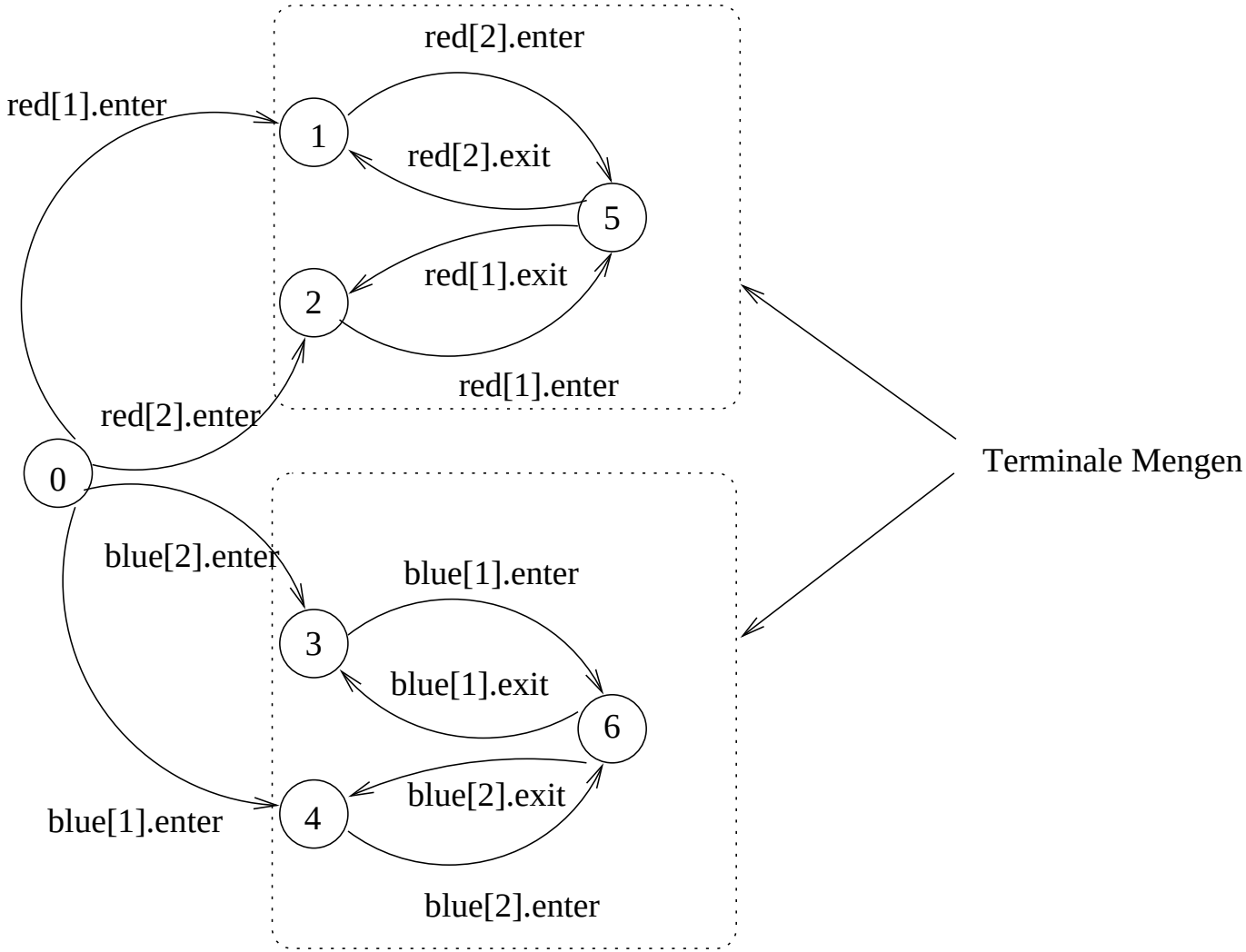
1. Möglichkeit: Der Scheduler wählt nur rote oder nur blaue Autos.
   Das ist nicht realistisch.
2. Möglichkeit: Der Prozessor soll möglichst gut ausgelastet werden, d.h. im Beispiel möglichst viele (gleichfarbige) Autos auf die Brücke lassen.

Gestresstes Modell:

“enter“-Aktionen erhalten hohe Priorität.

$\|\mathrm{FULLBRIDGE} = (\mathrm{SYS}) << \{\mathrm{red[ID].enter}, \mathrm{blue[ID].enter}\}$.

LTS von FULLBRIDGE für N=2:



Fortschrittsanalyse für FULLBRIDGE:

progress BLUECROSS = {blue[ID].enter}.
progress REDCROSS = {red[ID].enter}.

Beide Fortschrittseigenschaften werden von FULLBRIDGE **nicht** erfüllt.

FULLBRIDGE $\not\models$ BLUECROSS
" $\not\models$ REDCROSS

Überarbeitung des Modells:

1. Versuch:

Die Brücke lässt nur dann rote Autos auffahren, wenn kein blaues Auto auf der Brücke ist *und* wenn kein blaues Auto wartet. (Analog für blaue Autos!)

Ändere CAR in:

CAR = (request → enter → exit → CAR).

```
BRIDGE = BRIDGE[0][0][0][0],
BRIDGE[nr:T][nb:T][wr:T][wb:T] =
   (red[ID].request   -> BRIDGE[nr][nb][wr+1][wb]
   |when (nb==0 && wb==0)
       red[ID].enter  -> BRIDGE[nr+1][nb][wr-1][wb]
   |red[ID].exit      -> BRIDGE[nr-1][nb][wr][wb]
   |blue[ID].request  -> BRIDGE[nr][nb][wr][wb+1]
   |when (nr==0 && wr==0)
       blue[ID].enter -> BRIDGE[nr][nb+1][wr][wb-1]
   |blue[ID].exit     -> BRIDGE[nr][nb-1][wr][wb]
   ).
```

Checken der 1. Revision von SYS:

Jetzt besitzt SYS = (CARS ‖ BRIDGE) ein DEADLOCK.

Ein minimaler Ablauf dahin ist (bei N = 2):

```
red[1].request  red[2].request  blue[1].request  blue[2].request
```

2. Versuch:

Wie Versuch 1, jedoch darf ein Auto auffahren, wenn seine Farbe an der Reihe ist, auch wenn andersfarbige Autos warten.

```
const True = 1
const False = 0
range B = False..True //bt=True: blue turn, bt=False: red turn
BRIDGE = BRIDGE[0][0][0][0][True],
BRIDGE[nr:T][nb:T][wr:T][wb:T][bt:B] =
   (red[ID].request   -> BRIDGE[nr][nb][wr+1][wb][bt]
   |when (nb==0 && (wb==0 || !bt))
       red[ID].enter  -> BRIDGE[nr+1][nb][wr-1][wb][bt]
   |red[ID].exit      -> BRIDGE[nr-1][nb][wr][wb][True]
   |blue[ID].request  -> BRIDGE[nr][nb][wr][wb+1][bt]
   |when (nr==0 && (wr==0 || bt))
       blue[ID].enter -> BRIDGE[nr][nb+1][wr][wb-1][bt]
   |blue[ID].exit     -> BRIDGE[nr][nb-1][wr][wb][False]
   ).
```

Beachte:

Auto darf auch dann auffahren, wenn seine Farbe nicht an der Reihe ist aber kein andersfarbiges Auto wartet.

Möglicher Ablauf von SYS = (CARS ‖ BRIDGE)
(Blau ist zu Beginn an der Reihe):

```
red[1].request
blue[1].request
blue[1].enter
blue[2].request
blue[2].enter
blue[1].exit
blue[1].request
blue[2].exit
blue[2].request
red[1].enter
...
```

jetzt ist bt = FALSE

weil

Checken der 2. Revision von SYS:

Das neue SYS erfüllt alle Sicherheits- und Fortschrittseigenschaften und ist Deadlock-frei.

Fortschrittsanalyse für FULLBRIDGE nach 2. Revision von SYS:

progress BLUECROSS $= \{$blue[ID].enter$\}$.
progress REDCROSS $= \{$red[ID].enter$\}$.

Beide Fortschrittseigenschaften werden von FULLBRIDGE erfüllt.

Weitere unfaire Abläufe nicht realistisch $\Longrightarrow$ Implementieren!

Erneute Implementierung:

Diese erfolgt gemäß des revidierten Systemmodells.

```
class FairBridge extends Bridge {
   private int nred  = 0;    // count of red cars on the bridge
   private int nblue = 0;    // count of blue cars on the bridge
   private int waitblue = 0; // count of waiting blue cars
   private int waitred  = 0; // count of waiting red cars
   private boolean blueturn = true;

   synchronized void redEnter() throws InterruptedException {
      waitred++;
      while (nblue>0 || (waitblue>0 && blueturn)) wait();
      waitred--;
      nred++;
   }

   synchronized void redExit() {
      nred--;
      blueturn = true;
      if (nred == 0) notifyAll();
   }
```

```
    synchronized void blueEnter() throws InterruptedException {
        waitblue++;
        while (nred>0 || (waitred>0 && !blueturn)) wait();
        waitblue--;
        nblue++;
    }

    synchronized void blueExit() {
        nblue--;
        blueturn = false;
        if (nblue == 0) notifyAll();
    }
}
```